Probabilistic Prediction of the QoS of Service Orchestrations: A Truly Compositional Approach*

Leonardo Bartoloni, Antonio Brogi, and Ahmad Ibrahim

Department of Computer Science, University of Pisa, Italy
{bartolon,brogi,ahmad}@di.unipi.it

Abstract. The ability to a priori predict the QoS of a service orchestration is of pivotal importance for both the design of service compositions and the definition of their SLAs. QoS prediction is challenging because the results of service invocations is not known a priori. In this paper we present an algorithm to probabilistically predict the QoS of a WS-BPEL service orchestration. Our algorithm employs Monte Carlo simulations and it improves previous approaches by coping with complex dependency structures, unbound loops, fault handling, and unresponded service invocations.

Keywords: QoS prediction, service orchestration, WS-BPEL, Monte Carlo method.

1 Introduction

Quality of Service (QoS) of a service orchestration depend on the QoS of services it invokes. When selecting and composing various services together, the designer of an orchestrator has to consider whether the desired composition yields an overall QoS level which is acceptable for the application. In order to predict QoS two characteristics of service orchestration must be considered:

- *Different results of service invocations.* Each invoked service can return a successful reply, a fault notification, or even no reply at all. If a fault is returned, a fault handling routine will be executed instead of the normal control flow. If no reply is received, the orchestrator may wait forever for a reply (unless some parallel branch throws a fault). In either case, the resulting QoS of the composition differs from the case of successful invocation.
- *Non-determinism in the workflow.* Different runs of the same application can have different QoS values just because the orchestration control flow is non-deterministic due to two reasons. Firstly, different runs of the orchestration can get different service invocation results (success/fault/no reply). It is worth noting that a service is not always faulty or successful, rather it has a certain probability of being successful (as guaranteed in its SLA). Secondly,

* Work partly supported by the EU-FP7-ICT-610531 SeaClouds project.

X. Franch et al. (Eds.): ICSOC 2014, LNCS 8831, pp. 378–385, 2014.

alternative and iterative control flow structures (if/else and loops) depend on input data which may differ in different runs. This leads, for instance, to different numbers of loop iterations or to different branches executed in a if/else structure. Moreover certain QoS properties of invoked services can vary from one run to another (e.g., response time).

The objective of this paper is to present an algorithm to probabilistically predict the QoS of a workflow defining a service orchestration. The inputs of the algorithm are a WS-BPEL [1] workflow, and probability distributions for the QoS properties of the services used as well as for branch guard evaluations. The output of the algorithm is a probability distribution for the QoS properties of the orchestration. We represent distributions (both input and output) as *sampling functions* due to which not only we can compute average/expected values but also many other statistical properties (e.g., standard deviation or the probability of QoS not respecting a target SLA) by using the Monte Carlo method [2]. Our method provides a more accurate representation than traditional sequential and parallel decomposition, by using a different pair of basic composition functions which can model more suitably arbitrary dependency structures, unbound loops and fault handling in a compositional way. Furthermore, our method improves previous work by providing more accurate predictions by modeling a certain degree of correlation between parallel branches.

2 Related Work

Various approaches (e.g., [3–9]) have been proposed to determine the QoS of service compositions.

Cardoso [3] presented a mathematical model and an algorithm to compute the QoS of a workflow composition. He iteratively reduces the workflow by removing parallel, sequence, alternative and looping structures according to a set of reduction rules, until only one activity remains. However, some workflow complex dependencies cannot be decomposed into parallel or sequence, as shown in [9]. This kind of approach has been adopted also by others [5, 7, 8], some of whom (e.g., [4]) tried to overcome such limitation by defining more reduction patterns.

Mukherjee et al. [6,9] presented a algorithm to estimate the QoS of WS-BPEL compositions. They convert a WS-BPEL workflow into an activity dependency graph, and assign probabilities of being executed to each activity. In their framework it is possible to treat any arbitrary complex dependency structure as well as *fault* driven flow control. However, they do not consider correlation between activities which do not have a direct dependency, and this in some cases can yield a wrong result.

Zheng et al. [8] focused on QoS estimation for compositions represented by service graphs. In their approach however they only marginally deal with parallelism, by not considering arbitrary synchronization links (i.e., they restrict to cases in which is possible to decompose *flow*-like structures into parallel and sequences, as in [3]), and they do not take into account fault handling. Moreover,

they need to fix an upper bound to the number of iterations for cycles, in order to allow decomposition into acyclic graph. They also assume that service invocations are deterministic, namely services are always successful and their QoS is not changing from one run to another.

To the best of our knowledge all previous approaches require to know a priori the exact number of iterations, or at least an upper bound for each loop in order to estimate QoS values. Also, other approaches rarely take fault handling into account, and never deal with non-responding services.

3 Determine the QoS of a Service Orchestration

In this section we introduce our algorithm to provide a QoS estimate for a service orchestration based on the QoS of the services it invokes. Our input workflows can contain any arbitrary dependency structure (i.e., not only for parallel and sequential execution patterns), fault handling, unbound loops and can preserves correlation, for example in diamond dependencies.

Our algorithm uses a structural recursive function that associates each WS-BPEL activity with a *cost* structure. This cost structure is a tuple of metadata chosen accordingly to the QoS values we want to compute. The *cost* structure has to carry enough information to allow computation of QoS values and allow composing it with other costs using the standard WS-BPEL constructs, i.e. it needs to have a composition function for each WS-BPEL construct. Later we will show that it is possible to write a composition function for most of WS-BPEL composition constructs by only requiring two basic operations on the *cost* data type. The first is the compositor for independent parallel execution of two activities. Suppose we have two activities `A` and `B`, we assume to be able to compute the cost of executing both in parallel only knowing the cost of those activities, by using a given function `Both`. The second compositor is the one we use to resolve dependency. If a WS-BPEL construct of `A` and `B` introduces some dependency/synchronization between the two activities, namely we suppose that it forces the activity `B` to start after completion of `A`, we will need to adjust the cost of `B` to take into account the dependence introduced by the composition structure, and we suppose to be able to do it from the costs of `A` and `B` by using a given operation `Delay`[1]. For example in our model the `Sequence(A,B)` construct is decomposed into a parallel execution of the independent activity `A` and the activity `B` synchronized after `A`, as such its cost can be written, in absence of faults, as:

$$\frac{\mathrm{Cost}(A) = cA \qquad \mathrm{Cost}(B) = cB}{\mathrm{Cost}(\mathrm{Sequence}(A, B)) = \mathrm{Both}(cA, \mathrm{Delay}(cB, cA))}$$

This is similar to what has been done in previous approaches (e.g., [3]) in which the `Flow` dependency graph is decomposed into parallel and sequence

[1] We use `Delay` as function name because in most cases this affects only time-based properties of the dependent activity, such as completion time.

compositions. By choosing `Both` and `Delay` as basic composition operators however we can define cost composition functions for any dependence structure, while the parallel and sequence decomposition fails for a significantly wide range of dependency graph allowed by the WS-BPEL `Flow` construct [9].

Because of the definition it can be verified that functions `Both` and `Delay` need to respect the following properties:

- `Both` is commutative, i.e. $\forall a, b.\, \mathrm{Both}(a, b) = \mathrm{Both}(b, a)$
- `Both` is associative, i.e. $\forall a, b, c.\, \mathrm{Both}(a, \mathrm{Both}(b, c)) = \mathrm{Both}(\mathrm{Both}(a, b), c)$
- `Delay` is associative, i.e. $\forall a, b, c.\, \mathrm{Delay}(a, \mathrm{Delay}(b, c)) = \mathrm{Delay}(\mathrm{Delay}(a, b), c)$
- `Delay` is right-distributive over `Both`, i.e. $\forall a, b, c.\, \mathrm{Delay}(\mathrm{Both}(a, b), c)) = \mathrm{Both}(\mathrm{Delay}(a, c), \mathrm{Delay}(b, c))$

We also explicitly name a neutral element `Zero` (i.e. : `Both(A,Zero) = A` and `Delay(A,Zero) = A`) which can be useful for example to define the `All` function, which extend the `Both` function to any number of parameters:

$$\frac{}{\mathrm{All}([\,]) = \mathrm{Zero}} \qquad \frac{\mathrm{All}(t) = tc}{\mathrm{All}(h :: t) = \mathrm{Both}(h, tc)}$$

3.1 Control Flow Trimming

In WS-BPEL there are two control flow mechanism that will ultimately result in some activities not being executed: Explicit control flow (`IfThenElse` statements, iterations, and synchronization `<link>` status) and faults management. To effectively resolve such control flow structures and exclude from computation costs of activities which are not executed, we require to associate additional metadata to an activity:

- To resolve explicit control flow we assume an *environment* holding the synchronization `<link>` status and variable values. We restrict to Boolean variables in order to keep the size of the environment finite, and thus computable.
- To resolve fault handling we compute also the *outcome* of an activity, i.e., whether an activity is successfully executed or not. We identify three different outcomes for an activity: the `Success` outcome, which result in execution of consequent activities and skipping eventual *fault handlers*, the `Fault` outcome, which on the opposite will result in skipping consequent activities but executing *fault handlers*, the `Stuck` outcome is assigned to activities where the orchestrator waits for a service which failed to provide a response.

3.2 Statistical Non-determinism

It is not possible to define a deterministic function that given an activity and an input *environment* yields its *outcome*, its *cost* and the modified *environment* because:

- Outcome and cost of `Invoke` activities are in general non-deterministic, because they depend on external services.

– Data dependent control flow can not be evaluated exactly, because data values are unknown.

We can however define an `Eval` function which computes a distribution on the *outcome*, *cost* and output *environment* for a given activity and a given status of input *environment*. Many models can be chosen to represent a distribution. For simplicity we choose sampling functions for this purpose. Sampling functions are algorithm that extracts random values according to the distribution being represented, which can be used in a Monte Carlo simulation to retrieve probabilities and expected values. A structural recursive definition of such `Eval` function can be given by exploiting the monadic property of distributions, i.e., if an expression contains some variable whose distribution is known the distribution for the value of the expression can be computed (by integrating the variable). For sampling functions this means that it is possible to generate samples for an expression that contains a random variable for which a sampling function is available, which can be done by sampling the variable first then replacing its value inside the expression.

To give a grasp of the algorithm we give an example of the `Eval` function for the `Scope` construct (written using F# [10] programming language). As the expression depends on two subactivities, it recursively compute the sampling function for needed subactivities, then evaluate it when needed. Here the `Scope(A,H)` expression represents a scope activity with inner activity `A` and fault handler `H`, and generator is an entropy source to be used for sampling:

```
let Eval (Scope(A,H)) env =
    fun generator ->
        let aSamplingFun = Eval A env
        let newEnv,outcome,cost = asamplingFun generator
        if outcome = Fault then
            let hSamplingFun = Eval newEnv H
            let newerEnv,outcome, newCost = hSamplingFun generator
            newerEnv,outcome,Both(cost,Delay(newCost,cost))
        else
            newEnv,outcome,cost
```

From the flow analysis point of view the `Scope` activity is very similar to a `Sequence`, except that while `Sequence` executes the second activity only if the first is successful, in `Scope` the fault handler is executed only when the first yields a `Fault`. For external invocations we expect to have a sampling function describing the service, which can be written according to the service's QoS. Note that if the service has a WS-BPEL description, its sampling function can be computed in the same way with this algorithm.

```
let Eval (Invoke(s)) env =
    s.getSamplingFunction()
```

As explicit control flow construct, we implemented deterministic `IfThenElse`, whose sampling function evaluates the guard on the environment and then delegates sampling to either of branches. The transition/join conditions in the `Flow` model are implemented in a similar fashion. For the `While` loop construct the body is sampled until either the guard yields false for the output environment or a `Fault` or `Stuck` result are reached. In this case too we assume the guard

evaluation to be deterministic. Since we do not allow random branching we introduce random Boolean variable assignment (`OpaqueAssign`). A random branching can be emulated by replacing it with a `Sequence` of random variable assignment followed by the branch instruction. We purposely do not allow random branching and random transition/join condition evaluation for two reasons: first it simplifies the model by keeping only one construct which introduces randomness, secondly it makes clear to the user when conditions are correlated and when they are not. We also allow a deterministic `Assign` instruction to perform evaluation of Boolean expressions which are not immediately bound to a branch instruction.

For `Flow` we sort all activities according to the link dependencies, then for each of them we recursively compute sampling functions and generate samples for each activity outcome, cost and output environment. We store the outcome and the cost delayed by the cost of all dependencies, evaluate transition conditions, which are deterministic, and store link statuses. This allow us to skip all activities where one of the dependencies has a `Stuck` or `Fault` outcome, or whose join condition is not satisfied. We assume that there is no race condition on variables, i.e. if the same variable is used by two activities the two activities have a dependency relation (i.e. one depends on the other or vice versa), thus we only keep track of one environment. The `Flow` activity *outcome* will be successful if all activities inside it are successful, will be a `Fault` if at least one of the activities is faulty, `Stuck` otherwise. The *cost* is computed by merging together all delayed costs for inner activities using `Both`/`All`, since the `Flow` construct encodes parallel execution. The *environment* is the one resulting after executing all activities.

4 Example

To illustrate our approach, we consider a bank customer loan request example (Figure 1), which is variation of the well-known WS-BPEL loan example [1]. We want to estimate values for the Reliability, amortized expense for successful execution and average response time of this composition. Let us assume for the loan example the distribution of variable assignments and invoked services QoS shown in Table 1.

Table 1. Input distributions

	True	False
bigAmount	50%	50%
highRisk	60%	40%

(a) Control Flow

	Success	Fault	Stuck
0.1$, 1 sec	79%	-	-
0.1$, 2 sec	20%	-	-
0.1$, 0 sec	-	-	1%

(b) Risk Assessment

	Success	Fault	Stuck
5$, 10 min	30%	-	-
10$, 20 min	35%	-	-
15$, 30 min	20%	-	-
0$, 5 min	-	15%	-

(c) Approval

The algorithm will start by evaluating the *cost* and *outcome* for the outermost `Flow` activity and computes *delayed costs* for the activities in the flow, and then sums them with the `All` compositor. Table 2 summarizes six runs of the `Eval`

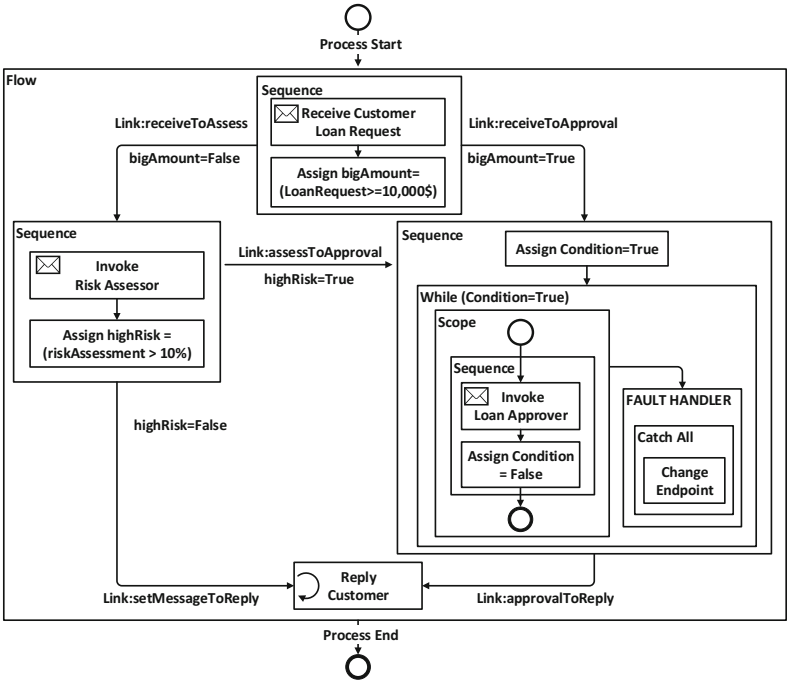

Fig. 1. Loan Request Example

function on the loan request example. To estimate the required QoS properties, we will perform a Monte Carlo sampling. Reliability can be determined by computing the expectation of `successCount`. *Amortized expense* and *average response time* are divided by reliability to normalize them with respect to the number of successful executions.

Table 2. Total cost for different runs of the loan example

bigAmount	highRisk	Risk Assessment	Approval(s)	Composition
True		Success (Zero)	Fault (0$, 5 min); Success (5$, 10 min)	Success (5$, 15 min)
False	False	Success (0.1$, 2 sec)		Success (0.1$, 2 sec)
False	True	Success (0.1$, 1 sec)	Success (15$, 30 min)	Success (15.1$, 181 sec)
True		Success (Zero)	Fault (0$, 5 min); Success (10$, 20 min)	Success (10$, 25 min)
True		Success (Zero)	Success (15$, 30 min)	Success (15$, 30 min)
False		Stuck (0.1$, 0)		Stuck (0.1$, 0)

By computing the above values for the samples of Table 2 we get:

$\texttt{expectedSuccessfulTime} = \frac{1}{6} \cdot (15 \cdot 60 + 2 + 181 + 25 \cdot 60 + 30 \cdot 60 + 0) = \frac{6003}{6}$ sec
$\texttt{expectedExpense} = \frac{1}{6} \cdot (5 + 0.1 + 15.1 + 10 + 15 + 0.1) = \frac{45.3}{6}$$
$\texttt{reliability} = \frac{5}{6} = 83\%$
$\texttt{amortizedExpense} = \frac{45.3}{5} = 9.06$$
$\texttt{averageResponseTime} = \frac{6003}{5} = 1200.6 \text{ sec} = 20 \text{ min } 0.6 \text{ sec}$

5 Conclusions

In this paper we have presented a novel approach to probabilistically predict the QoS of service orchestrations. Our algorithm improves previous approaches by coping with complex dependency structures, unbound loops, fault handling, and unresponded service invocations. Our algorithm can be fruitfully exploited both to probabilistically predict QoS values before defining the SLA of an orchestration and to compare the effect of substituting one or more endpoints (viz., remote services).

We see different possible directions for future work. One of them is to extend our approach to model some other WS-BPEL constructs that we have not discussed in this paper, like `Pick` and `EventHandlers`. Another possible extension could be to allow for cases in which *no information at all* (not even a branch execution probability) is available for flow control structures. Similarly the *uncorrelated samples* restriction imposed on invocations and assignments should be relaxed. We would also like to be able to specify some degree of correlation between consecutive samples (e.g., if a service invocations yields a fault because it is "down for maintenance" we should increase the probability of getting the same fault in the next invocation).

References

1. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al.: Web services business process execution language version 2.0. OASIS standard 11 (2007)
2. Dunn, W.L., Shultis, J.K.: Exploring Monte Carlo Methods. Elsevier (2011)
3. Cardoso, A.J.S.: Quality of service and semantic composition of workflows. PhD thesis, Univ. of Georgia (2002)
4. Jaeger, M., Rojec-Goldmann, G., Muhl, G.: QoS aggregation for web service composition using workflow patterns. In: Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference, EDOC, pp. 149–159 (2004)
5. Ben Mabrouk, N., Beauche, S., Kuznetsova, E., Georgantas, N., Issarny, V.: QoS-Aware service composition in dynamic service oriented environments. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 123–142. Springer, Heidelberg (2009)
6. Mukherjee, D., Jalote, P., Gowri Nanda, M.: Determining QoS of WS-BPEL compositions. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 378–393. Springer, Heidelberg (2008)
7. Wang, H., Sun, H., Yu, Q.: Reliable service composition via automatic QoS prediction. In: IEEE International Conference on Services Computing (SCC), pp. 200–207 (2013)
8. Zheng, H., Zhao, W., Yang, J., Bouguettaya, A.: Qos analysis for web service compositions with complex structures. IEEE Transactions on Services Computing 6, 373–386 (2013)
9. Mukherjee, D.: QOS IN WS-BPEL PROCESSES. Master's thesis, Indian Institute of Technology, Delhi (2008)
10. Syme, D., Granicz, A., Cisternino, A.: Expert F# 3.0, 3rd edn. Apress, Berkeley (2012)